C++ guidelines

ИТМО 2018

- **Вся презентация сделана по принципу «содержимое важнее эстетического удовольствия»**
- **Если вы ожидаете красивую презентацию или у вас приступ перфекционизма – покиньте аудиторию**

- **Допускаются любые нарушения рекомендаций, если это улучшает читаемость.**
- Основная цель рекомендаций — улучшение читаемости и, следовательно, ясности и лёгкости поддержки, а также общего качества кода. Невозможно дать рекомендации на все случаи жизни, поэтому программист должен мыслить гибко

- **Правила могут быть нарушены, если против них есть персональные возражения.**
- Это попытка создать набор общих рекомендаций, не навязывая всем единый стиль. Опытные программисты обычно всё равно подгоняют стиль под себя. Подобный список рекомендаций, имеющийся под рукой (или хотя бы требование ознакомиться с ним), обычно заставляет людей задумываться о стиле программирования и оценке их собственных практик в этой области.

  С другой стороны, новички и неопытные программисты обычно используют рекомендации по стилю для лучшего понимания жаргона программистов.
- Это вам на будущее, в рамках нашего курса это правило работает только для преподавателей ☺

Соглашения об именовании

- **Имена, представляющие типы, должны быть обязательно написаны в смешанном регистре (CamelCase), начиная с верхнего.**

Line,
SavingsAccount

- **Имена переменных должны быть записаны в смешанном регистре, начиная с нижнего.**

line,
savingsAccount

- **Именованные константы (включая значения перечислений) должны быть записаны в верхнем регистре с нижним подчёркиванием в качестве разделителя.**

MAX_ITERATIONS, COLOR_RED, PI

- **Названия методов и функций должны быть глаголами, быть записанными в смешанном регистре и начинаться с нижнего.**

getName(),
computeTotalWidth()

- **Названия пространств имён следует записывать в нижнем регистре.**

model::analyzer,
io::iomanager,
common::math::geometry

- **Следует называть имена типов в шаблонах одной заглавной буквой.**

```
template<class T> ...
template<class C, class D> ...
```

Позволяет выделить имена шаблонов среди других используемых имён.

- **Аббревиатуры и сокращения в именах должны записываться в нижнем регистре.**

```
exportHtmlSource(); // НЕЛЬЗЯ: exportHTMLSource();
openDvdPlayer(); // НЕЛЬЗЯ: openDVDPlayer();
```

Когда имя связано с другим, читаемость снижается; слово, следующее за аббревиатурой, не выделяется так, как следовало бы

- **Глобальные переменные всегда следует использовать с оператором разрешения области видимости (::).**

::mainWindow.open(),

::applicationContext.getName()

Следует избегать использования глобальных переменных.

Рекомендация

- **Членам класса с модификатором** private **следует добавлять префикс m**

```
class SomeClass {
        private:
                int mLength;
}
```

```
void setDepth (int depth)
{
        mDepth = depth;
}
```

Есть много вариантов выделения private полей, но выделять нужно и в этом все сходятся

- **Настраиваемым переменным следует давать то же имя, что и у их типа.**

```
void setTopic(Topic* topic) // НЕЛЬЗЯ: void setTopic(Topic* value)
                            // НЕЛЬЗЯ: void setTopic(Topic* aTopic)
                            // НЕЛЬЗЯ: void setTopic(Topic* t)

void connect(Database* database) // НЕЛЬЗЯ: void connect(Database* db)
                                 // НЕЛЬЗЯ: void connect (Database* oracleDB)
```

Сокращайте сложность путём уменьшения числа используемых терминов и имён. Также упрощает распознавание типа просто по имени переменной.

Если по какой-то причине эта рекомендация кажется неподходящей, это означает, что имя типа выбрано неверно.

•

Не являющиеся настраиваемыми переменные могут быть названы по их назначению и типу:

Point startingPoint, centerPoint;
Name loginName;

- **Все имена следует записывать по-английски.**

fileName; *// НЕ РЕКОМЕНДУЕТСЯ: imyaFayla*

Английский наиболее предпочтителен для интернациональной разработки..

- **Переменные, имеющие большую область видимости, следует называть длинными именами, имеющие небольшую область видимости — короткими.**

Имена временных переменных, использующихся для хранения временных значений или индексов, лучше всего делать короткими. Программист, читающий такие переменные, должен иметь возможность предположить, что их значения не используются за пределами нескольких строк кода. Обычно это переменные *i*, *j*, *k*, *l*, *m*, *n* (для целых), а также *c* и *d* (для символов).

- **Имена объектов не указываются явно, следует избегать указания названий объектов в именах методов.**

line.getLength(); *// НЕ РЕКОМЕНДУЕТСЯ: line.getLineLength();*

Второй вариант смотрится вполне естественно в объявлении класса, но совершенно избыточен при использовании, как это и показано в примере.

Особые правила именования

- **Слова** get/set **должны быть использованы везде, где осуществляется прямой доступ к атрибуту.**

```
employee.getName();
employee.setName(name);
matrix.getElement(2, 4);
matrix.setElement(2, 4, value);
```

- **Слово** compute **может быть использовано в методах, вычисляющих что-либо.**

```
valueSet->computeAverage();
matrix->computeInverse()
```

- Дайте читающему сразу понять, что это времязатратная операция.

- **Слово** find **может быть использовано в методах, осуществляющих какой-либо поиск.**

```
vertex.findNearestVertex();
matrix.findMinElement();
```

- Дайте читающему сразу понять, что это простой метод поиска, не требующий больших вычислений.

- **Слово** initialize **может быть использовано там, где объект или сущность инициализируется.**

```
printer.initializeFontSet();
```

- Следует отдавать предпочтение американскому варианту initiali**z**e, нежели британскому initiali**s**e. Следует избегать сокращения init.

- **Множественное число следует использовать для представления наборов (коллекций) объектов.**

```
vector<Point> points;
int values[];
```

Однако порой даже лучше указать тип в переменной

```
map<int, std::string> mapNumber2Family;
vector<Mark> vectorMarks;
```

- **Переменным-итераторам следует давать имена** i, j, k **и т. д.**

```
for (int i = 0; i < nTables); i++)
{ ... }
for (vector<MyClass>::iterator i = list.begin(); i != list.end(); i++)
{
        Element element = *i; ...
}
```

- Обозначение взято из математики, где оно является установившимся соглашением для обозначения итераторов.

  Переменные с именами *j*, *k* и т. д. рекомендуется использовать только во вложенных циклах.

- **Префикс** is **следует использовать только для булевых (логических) переменных и методов.**

isSet, isVisible, isFinished, isFound, isOpen

- Использование этого префикса избавляет от таких имён,
  как *status* или *flag*. *isStatus* или *isFlag* просто не подходят, и программист вынужден выбирать более осмысленные имена.

- В некоторых ситуациях префикс *is* лучше заменить на другой: *has*, *can* или *should*:

```
bool hasLicense();
bool canEvaluate();
bool shouldSort();
```

- **Симметричные имена должны использоваться для соответствующих операций.**

get/set, add/remove, create/destroy,
- start/stop, insert/delete,

increment/decrement, old/new, begin/end, first/last, up/down, min/max, next/previous, old/new, open/close, show/hide, suspend/resume, и т. д.

- **Следует избегать сокращений в именах.**
-

computeAverage*()*; *// НЕЛЬЗЯ: compAvg();*

- Рассмотрим два вида слов. Первые — обычные слова, перечисляемые в словарях, которые нельзя сокращать. Никогда не сокращайте:

cmd вместо command
cp вместо copy
pt вместо point
comp вместо compute
init вместо initialize и т. д.

- Второй вид — слова, специфичные для какой-либо области, которые известны по своему сокращению/аббревиатуре. Их следует записывать сокращённо. Никогда не пишите:

HypertextMarkupLanguage вместо html
CentralProcessingUnit вместо cpu и т. д.

- **Следует избегать дополнительного именования ссылок.**

```
Line& line = …;  // НЕЛЬЗЯ: Line& rLine;
                 // НЕЛЬЗЯ : Line& lineRef;
```

- **Старайтесь избегать дополнительного именования указателей.**

```
Line* line; // НЕ РЕКОМЕНДУЕТСЯ: Line* pLine;
            // НЕ РЕКОМЕНДУЕТСЯ: Line* linePtr;
```

- **Классам исключений следует присваивать суффикс** Exception**.**

```
class AccessException
{
        :
};
```

- Классы исключений в действительности не являются частью архитектуры программ, и такое именование отделяет их от других классов.

- **Функциям (методам, возвращающим какие-либо значения) следует давать имена в зависимости от того, что они возвращают, а процедурам — в зависимости от того, что они выполняют (методы void).**
- Улучшайте читаемость. Такое именование даёт понять, что метод делает, а что нет, а также избавляет код от потенциальных побочных эффектов.

Файлы

- **Заголовочным файлам C++ следует давать расширение** .h
- **Если в заголовочном файле имеется код (нужно для шаблонов), расширение должно быть .hpp**
- **Файлы исходных кодов могут иметь расширения** .cpp **либо** .c++**.**

MyClass.cpp, MyClass.h

- **Класс следует объявлять в заголовочном файле и определять (реализовывать) в файле исходного кода, имена файлов совпадают с именем класса.**

MyClass.h, MyClass.cpp

- Облегчает поиск связанных с классом файлов. Очевидное исключение — шаблонные классы, которые должны быть объявлены и определены в заголовочном файле.

- **Все определения должны находиться в файлах исходного кода.**

```
class MyClass
{
public:
        int getValue () {return value_;} // НЕЛЬЗЯ!
}
```

- Заголовочные файлы объявляют интерфейс, файлы исходного кода его реализовывают. Если программисту необходимо найти реализацию, он должен быть уверен, что найдёт её именно в файле исходного кода.

- **Содержимое файлов не должно превышать 80 колонок.**

- 80 колонок — широко распространённое разрешение для редакторов, эмуляторов терминалов, принтеров и отладчиков; файлы передаются между различными людьми, поэтому нужно придерживаться этих ограничений. Уместная разбивка строк улучшает читаемость при совместной работе над исходным кодом.
- На деле в небольших проектах люди лояльны, и если у всех широкие мониторы главным требованием является то, чтобы строки умещались на экран

- **Заголовочные файлы должны содержать защиту от вложенного включения.**

- **#pragma once; если компилятор позволяет**

```
#ifndef COM_COMPANY_MODULE_CLASSNAME_H
#define COM_COMPANY_MODULE_CLASSNAME_H
:
#endif // COM_COMPANY_MODULE_CLASSNAME_H
```

- Конструкция позволяет избегать ошибок компиляции. Это соглашение позволяет увидеть положение файла в структуре проекта и предотвращает конфликты имён.

- **Директивы включения следует сортировать (по месту в иерархии системы, ниже уровень — выше позиция) и группировать. Оставляйте пустую строку между группами.**

```
#include <fstream>
#include <iomanip>

#include <qt/qbutton.h>
#include <qt/qtextfield.h>

#include "com/company/ui/PropertiesDialog.h"
#include "com/company/ui/MainWindow.h"
```

- **Директивы включения должны располагаться только в начале файла.**

  .

- **Never Use using In a Header File**

. This causes the name space you are using to be pulled into the namespace of the header file.

- **Use "" For Including Local Files**

. ... <> is reserved for system includes.

```
// Bad Idea. Requires extra -I directives to the compiler
// and goes against standards
#include <string>
#include <includes/MyHeader.hpp>

// Worse Idea
// requires potentially even more specific -I directives and
// makes code more difficult to package and distribute
#include <string>
#include <MyHeader.hpp>


// Good Idea
// requires no extra params and notifies the user that the file
// is a local file
#include <string>
#include "MyHeader.hpp"
```

- **Initialize Member Variables**

. ... with the member initializer list

```
class MyClass
{
public:
    MyClass(int value) : mValue(value)
    { }
private:
   int mValue;
};
```

- **Forward Declare when Possible**

. This is a proactive approach to simplify compilation time and rebuilding dependencies.

```
//good
// some header file
class MyClass;

void doSomething(const MyClass &);

//bad
// some header file
#include "MyClass.hpp"

void doSomething(const MyClass &);
```

Выражения

- **Локальные типы, используемые в одном файле, должны быть объявлены только в нём.**

Улучшает сокрытие информации.

- **Разделы**
  класса public**,** protected **и** private **должны быть отсортированы. Все разделы должны быть явно указаны.**

Сперва должен идти раздел *public*, что избавит желающих ознакомиться с классом от чтения
разделов *protected/private*.

- **Приведение типов должно быть явным. Никогда не полагайтесь на неявное приведение типов.**

```
floatValue = static_cast<float>(intValue); // НЕЛЬЗЯ: floatValue = intValue;
```

Этим программист показывает, что ему известно о различии типов, что смешение сделано намеренно.

- **Следует инициализировать переменные в месте их объявления.**

  Это даёт гарантию, что переменные пригодны для использования в любой момент времени. Но иногда нет возможности осуществить это:

```
int x, y, z;
getCenter(&x, &y, &z);
```

В этих случаях лучше оставить переменные неинициализированными, чем присваивать им какие-либо значения.

- **Не следует объявлять поля класса как** public**.**

  Эти переменные нарушают принципы сокрытия информации и инкапсуляции. Вместо этого используйте переменные с модификатором *private* и соответствующие функции доступа. Исключение — класс без поведения, практически структура данных (эквивалент структур языка C). В этом случае нет смысла скрывать эти переменные.

  Обратите внимание, что структуры в языке C++ оставлены только для совместимости с C; их использование ухудшает читаемость кода. Вместо структур используйте классы.

- **Символ указателя или ссылки в языке C++ следует ставить сразу после имени типа, а не с именем переменной.**

```
float* x; // НЕ РЕКОМЕНДУЕТСЯ: float *x;
int& y; // НЕ РЕКОМЕНДУЕТСЯ: int &y;
```

- То, что переменная — указатель или ссылка, относится скорее к её типу, а не к имени. Программисты на C часто используют другой подход, но в C++ лучше придерживаться этой рекомендации.

- **Переменные следует объявлять в как можно меньшей области видимости.**

- Это упрощает контроль над действием переменной и сторонними эффектами.

- **Нельзя включать в конструкцию for() выражения, не относящиеся к управлению циклом.**

```
sum = 0;
for (i = 0; i < 100; i++)
        sum += value[i];
```

-

```
// НЕЛЬЗЯ: for (i = 0, sum = 0; i < 100; i++)
sum += value[i];
```

- **Переменные, относящиеся к циклу, следует инициализировать непосредственно перед ним**

```
isDone = false;
while (!isDone) {
        …
}
```

- **Можно избегать циклов do-while.**

- Такие циклы хуже читаемы, поскольку условие описано после тела. Читающему придётся просмотреть весь цикл, чтобы понять его работу.

  Циклы do-while вообще не являются острой необходимостью. Любой такой цикл может быть заменён на цикл while или for.

  Меньшее число используемых конструкций улучшает читаемость

- **Следует избегать использования** break **и** continue **в циклах.**

- Такие выражения следует использовать только тогда, когда они повышают читаемость.

- **Для бесконечных циклов следует использовать форму** while (true) **.**
-

```
while (true) {
 :
}
for (;;) {
 // НЕТ!
:
}
while (1) {
 // НЕТ!
:
}
```

- **Строго избегайте сложных уловных выражений. Вместо этого вводите булевы переменные.**

```
bool isFinished = (elementNo < 0) || (elementNo > maxElement);
bool isRepeatedEntry = elementNo == lastElement;
if (isFinished || isRepeatedEntry)
{
        :
}
// NOT:
if ((elementNo < 0) || (elementNo > maxElement)|| elementNo ==
lastElement)
{
 :
}
```

- **Строго избегайте сложных уловных выражений. Вместо этого вводите булевы переменные.**

Задание булевых переменных для выражений приведёт к **самодокументированию** программы. Конструкцию будет легче читать, отлаживать и поддерживать.

- **Ожидаемую часть следует располагать в части** if**, исключение — в части** else**.**

```
bool isOk = readFile (fileName);
if (isOk)
{
 :
}
else
{
:
}.
```

- Это позволяет убедиться, что исключения не вносят неясности в нормальный ход выполнения. Важно для читаемости и производительности.

- **Условие следует размещать в отдельной строке.**
- **И вообще, нужно всегда ставить скобки, даже если у вас один оператор в условии / цикле**

```
if (isDone) // НЕ РЕКОМЕНДУЕТСЯ: if (isDone) doCleanup();
{
        doCleanup();
}
```

- Применяется для отладки.

- **Следует строго избегать исполнимых выражений в условиях.**

```
File* fileHandle = open(fileName, "w");
if (!fileHandle) {
        :
}
// НЕЛЬЗЯ:
if (!(fileHandle = open(fileName, "w")))
{ : }
```

- Исполняемые выражения в условиях усложняют читаемость..

- **Следует избегать «магических» чисел в коде. Числа, отличные от 0 или 1, следует объявлять как именованные константы.**

  Если число само по себе не имеет очевидного значения, читаемость улучшается путём введения именованной константы. Другой подход — создание метода, с помощью которого можно было бы осуществлять доступ к константе.

- **У функций нужно обязательно указывать тип возвращаемого значения.**

```
int getValue() // НЕЛЬЗЯ: getValue()
{ : }
```

Если это не указано явно, C++ считает, что возвращаемое значение имеет тип int. Никогда нельзя полагаться на это, поскольку такой способ может смутить программистов, не знакомых с ним.

- **Не следует использовать** goto**.**

- Этот оператор нарушает принципы структурного программирования. Следует использовать только в очень редких случаях (например, для выхода из глубоко вложенного цикла), когда иные варианты однозначно ухудшат читаемость.

- **Следует использовать «nullptr» вместо «NULL».**

- NULL является частью стандартной библиотеки C и устарело в C++.

- **Avoid raw memory access**

- Raw memory access, allocation and deallocation, are difficult to get correct in C++ without risking memory errors and leaks. C++11 provides tools to avoid these problems.

```
// Bad Idea
MyClass *myobj = new MyClass;
// ...
delete myobj;

// Good Idea
std::shared_ptr<MyClass> myobj = make_shared<MyClass>();
// ...
// myobj is automatically freed for you whenever it is no longer used.
```

- **Prefer pre-increment to post-increment**

- ... when it is semantically correct. Pre-increment is faster then post-increment because it does not require a copy of the object to be made.

```
// Bad Idea
for (int i = 0; i < 15; i++) {
    std::cout << i << std::endl;
}

// Good Idea
for (int i = 0; i < 15; ++i) {
std::cout << i << std::endl;
}
```

- **Const as much as possible**

- const tells the compiler that a variable or method is immutable. This helps the compiler optimize the code and helps the developer know if a function side effects. Also, using const & prevents the compiler from copying data unnecessarily.
- **Пример на следующем слайде**

```
// Bad Idea
class MyClass {
public:
    MyClass(std::string value) : mValue(value)
    { }
    std::string get_value() {
        return mValue;
    }
private:
    std::string mValue;
};

// Good Idea
class MyClass {
public:
    MyClass(const std::string& value) : mValue(value)
    { }
    std::string get_value() const {
       return mValue;
    }
private:
    std::string mValue;
};
```

- **Prefer Stack Operations to Heap Operations**

- Heap operations have performance penalties in mulithreaded environments on most platforms and can possibly lead to memory errors if not used carefully.
- Modern C++11 has special move operations which are designed to enhances the performance of stack based data by reducing or eliminating copies, which can bring even the single threaded case on par with heap based operations.

Оформление и комментарии

- **Основной отступ следует делать в четыре пробела.**

```
for (i = 0; i < nElements; i++)
{
    a[i] = 0;
}
```

- Отступ в один пробел достаточно мал, чтобы отражать логическую структуру кода. Отступ более 4 пробелов делает глубоко вложенный код нечитаемым и увеличивает вероятность того, что строки придётся разбивать. Широко распространены варианты в 2, 3 или 4 пробела; причём 2 и 4 — более широко.

- **Блоки кода следует оформлять так, как показано в примере 1 (рекомендуется) или в примере 2, но ни в коем случае не так, как показано в примере 3. Оформление функций и классов должно следовать примеру 2.**

```
while (!done)
{
    doSomething();
    done = moreToDo();
}
```

```
while (!done) {
    doSomething();
    done = moreToDo();
}
```

```
while (!done)
  {
    doSomething();
    done = moreToDo();
  }
```

- **Объявления классов следует оформлять следующим образом:**

```
class SomeClass : public BaseClass
{
public:
    ...
protected:
    ...
private:
    ...
}
```

- **Определения методов следует оформлять следующим образом:**

```
void someMethod()
{
    ...
}
```

- **Конструкцию** switch **следует оформлять следующим образом:**

```
switch (condition)
{
   case ABC :
      statements;
      // There is no "break"
    case DEF :
      statements;
      break;
    case XYZ :
       statements;
       break;
    default :
      statements;
      break;
}
```

Обратите внимание, что каждое слово *case* имеет отступ относительно всей конструкции, что помогает её выделить. Также обратите внимание на пробелы перед двоеточиями. Если где-то отсутствует ключевое слово *break*, то предупреждением об этом должен служить комментарий. Программисты часто забывают ставить это слово, поэтому случай нарочного его пропуска должен описываться специально.

— Операторы следует отбивать пробелами.
— После зарезервированных ключевых слов языка C++ следует ставить пробел.
— После запятых следует ставить пробелы.
— Двоеточия следует отбивать пробелами.
— После точек с запятой в цикле *for* следует ставить пробелы.

- **Логические блоки в коде следует отделять пустой строкой.**

```
Matrix4x4 matrix = new Matrix4x4();

double cosAngle = Math.cos(angle);
double sinAngle = Math.sin(angle);

matrix.setElement(1, 1, cosAngle);
matrix.setElement(1, 2, sinAngle);
matrix.setElement(2, 1, -sinAngle);
matrix.setElement(2, 2, cosAngle);

multiply(matrix);
```

- **Конструкцию** switch **следует оформлять следующим образом:**

```
switch (condition)
{
   case ABC :
      statements;
      // There is no "break"
    case DEF :
      statements;
      break;
    case XYZ :
       statements;
       break;
    default :
      statements;
      break;
}
```

Обратите внимание, что каждое слово *case* имеет отступ относительно всей конструкции, что помогает её выделить. Также обратите внимание на пробелы перед двоеточиями. Если где-то отсутствует ключевое слово *break*, то предупреждением об этом должен служить комментарий. Программисты часто забывают ставить это слово, поэтому случай нарочного его пропуска должен описываться специально.

- **Переменные в объявлениях можно выравнивать.**

```
AsciiFile*  file;
int         nPoints;
float       x, y;
```

- **Используйте выравнивание везде, где это улучшает читаемость.**

```
if      (a == lowValue)    compueSomething();
else if (a == mediumValue) computeSomethingElse();
else if (a == highValue)   computeSomethingElseYet();

value = (potential        * oilDensity)   / constant1 +
        (depth            * waterDensity) / constant2 +
        (zCoordinateValue * gasDensity)   / constant3;

minPosition     = computeDistance(min,     x, y, z);
averagePosition = computeDistance(average, x, y, z);

switch (value) {
  case PHASE_OIL   : strcpy(phase, "Oil");   break;
  case PHASE_WATER : strcpy(phase, "Water"); break;
  case PHASE_GAS   : strcpy(phase, "Gas");   break;
}
```

- **Сложный код, написанный с использованием хитрых ходов, следует не комментировать, а переписывать!**

- Следует делать как можно меньше комментариев, делая код самодокументируемым путём выбора правильных имён и создания ясной логической структуры.

- **Все комментарии следует писать на английском.**

- В интернациональной среде английский — предпочтительный язык.

- **Используйте // для всех комментариев, включая многострочные.**

- Если следовать этой рекомендации, многострочные комментарии /* */ можно использовать для отладки и иных целей.

  После // следует ставить пробел, а сам комментарий следует начинать писать с большой буквы завершать точкой.